intel

ARTICLE
REPRINT

AR-102

(19)

MASTER

July 1979

Designing Reliable Software for Automotive Applications

Barry Yarkoni and John Wharton
Intel Corp.

Designing Reliable Software for Automotive Applications

Barry Yarkoni and John Wharton
Intel Corp.

ABSTRACT

The use of microcomputers in automotive applications has placed new requirements on programmers. Programmers must protect their software against flaws in the hardware system in which it is operating. Various techniques for writing "fail-safe software" have been developed, and are discussed in this paper.

DESIGNERS MUST ACCEPT the fact that their hardware systems will be required to perform in an imperfect environment full of noise and glitches. While no expectation is made that a system perform perfectly under such conditions, most automotive applications will require the system to perform at a level consistent with vehicle safety and the integrity of related systems in the vehicle. For example, it might be acceptable for an emissions control system to occasionally deviate from specifications under unusual operating conditions, such as a loose ignition wire or broken insulation. However, it would be disastrous if a system generating fuel pulses became stuck with the injectors wide open.

Fulfilling these requirements in a manner consistent with low cost is a challenging problem for automotive systems designers. The techniques described in this paper are not a substitute for clean system design nor proper protection of the system from the ravages of environmental noise. After the designer has taken all possible steps to reduce the noise level in the system, these techniques may be applied to minimize or eliminate the consequences. Although the examples illustrate the use of an Intel 8048, the techniques described are general in nature.

FAILURE MODES

The failure mechanisms described here are not hard failures, but rather random occurances of noise spikes causing false inputs or changes in the state of various memory elements in the system. This latter phenomenon will be referred to as noise "hits".

Bad inputs can occur on any pin of any chip in the system. Typically, the most vulnerable points in the system are reset lines, control lines, and interrupts. Control line glitches can cause extraneous or erroneous opcode fetches if the program memory is external, which can subsequently result in "hits" as registers, accumulators and program counter assume various incorrect values.

Hits can occur anywhere. A bit in memory can change value, a register can change value, as can the stack pointer and program counter. I/O flip-flops can change state, and various flags such as interrupt enable can toggle. It becomes the programmer's task to design software that is as tolerant as possible of faults in any of these memory elements, provided they occur relatively infrequently.

Given the disturbing fact that any of these events can and will at least occasionally occur, what will be the effect on the system's operation? Some of the failure modes are obvious, while others obey Murphy's Law and occur at the worst possible time in the most undetectable way.

Probably the most common failure mode is the "stuck in tight loop" syndrome, generally due to the processor executing data. The program may execute data in tables, or the program counter can get out of sync with reality and execute the second byte of an instruction as if it were an opcode. Some single chip microcomputers have an I/O port which can also be used as a memory expansion bus (such as the Intel 8048). This feature makes it possible to execute data on the bus as instructions. The particular opcode(s) executed will be determined by the value of any inputs wired to the bus pins. This can in turn, cause a lock-up condition if the processor enters a "tight loop".

DETECTION & PREVENTION

There is no way that any microcomputer can detect that a lock-up has occured without the aid of either a dead-man timer (external hardware) or the use of an internal timer to generate a periodic interrupt. At the point the processor is interrupted, the program must check to see that certain checkpoints have been reached. This is done most easily if the program has been designed in a structured fashion. Using structured programming techniques, each

key section of a program will have only one entrance and exit point. At the exit points of blocks in the program, a counter can be incremented, and under normal circumstances the counter will be bumped fairly often. If, however, the interrupt routine detects that insufficient program execution has occurred, a recovery routine can be invoked.

Unfortunately, there is a small probability that this recovery scheme will not work in a particular instance. It is possible that the "tight loop" will be entered with interrupts disabled, or that data executed within the "tight loop" disables interrupts. In systems where such a failure can never be tolerated, it is then necessary to utilize an external deadman timer, brought in through a non-maskable interrupt or reset input. In other cases, the designer's goal is to reduce the probability that a "tight loop" will be entered and maximize the probability that it will be detected. The latter can be accomplished by minimizing the amount of time that the program spends with interrupts disabled. In addition, the existence of interrupt disable (DIS I) opcodes in the system should be minimized. While it may not be possible to eliminate the disable opcode from data tables (15H in the Intel 8048) it is certainly easy enough to disallow location of subroutines or loop points at memory locations ending in 15H. Following this practice eliminates the possibility of misinterpreting the second byte of JUMP and CALL instructions as disable instructions. In spite of these safeguards it is still prudent to periodically repeat mode control instructions such as "enable interrupt".

Another practice which can reduce the probability of executing a "tight loop" is the inclusion of as many "escape points" as possible throughout the program. All unused memory should be filled with NOP instructions (0's in the 8048) and strings of NOPS should be terminated with jumps to a recovery routine. Fig. 1 shows a convenient macro for filling to the end of a page with NOPS. Similarly, NOPS and JUMPS can be placed at the end of each data table with a small impact on overall code size. If the program has plenty of memory space to spare, tables can be expanded so that there are spaces between elements in the table. In the 8048, this causes any JUMPS or CALLS that do accidentally occur to branch to the first location of a 256 byte page. Jumps to a recovery routine can be placed at the beginning of each page. This feature is also included in Fig. 1. This technique can be carried to the extreme by placing a NOP after every 2 byte instruction whose second byte is the opcode of a JUMP or CALL, thus insuring that any false jumps or calls always go to the top of a 256 byte page, and then to a recovery routine.

Just as the programmer should take care that mode altering opcodes are minimized, there may also be "one-way" opcodes that can be just as disastrous. A one way opcode is an instruction that, once invoked, cannot be revoked without a

```
;=========================
;
;
; PGFILL - PAGE FILL MACRO
;
; FILLS ALL SPACE REMAINING ON PAGE WITH A SERIES
; OF NOPS FOLLOWED BY A JUMP TO RECOVR AT THE END
; OF THE PAGE.  IN ADDITION, A JUMP TO RECOVR IS
; INSERTED AT THE TOP OF THE NEXT PAGE OF MEMORY.
;
;=========================
;
PGFILL MACRO
       REPT    0FEH-(LOW $)
       NOP
       ENDM
$SAVE
$GEN
;
;      REMAINDER OF PAGE FILLED WITH NOP'S
;
       JMP     RECOVR
       JMP     RECOVR  ; BEGINNING OF NEXT PAGE
$RESTORE
       ENDM
```

Fig. 1 - Convenient macro for filling to the end of a page with NOPS

hardware reset. Many processors have such "one-way" operations to select I/O pin options. The existence of these opcodes should be minimized by the techniques already mentioned. An example in the 8048 is the ENTO CLK (75H) instruction which enables pin 1 as a clock output. Fig. 2 illustrates a code sequence which flags any use of location 75H for other than a NOP.

Memory and I/O bit "hits" are a common type of "hit" due to the large number of targets, especially in systems with external memory. Given that I/O bit "hits" will occur, it is possible to almost entirely eliminate their effects by several techniques, one of which is redundancy. By keeping an image of the state of all output bits in memory and periodically comparing the output to memory, a failure can be detected and a recovery procedure invoked.

Many systems cannot tolerate an output bit failure under any circumstances. This problem is best attacked by designing around the problem. One technique is to utilize software to generate duty cycle outputs AC coupled, band pass filtered, and rectified to control actuators such that only an output within a frequency range is able to cause an actuator to fire. In this manner, a pin stuck at one or zero cannot cause

```
;=========================
;
;
; ENSURE THAT NOP INSTRUCTION IS AT LOCATION X75H.
; GENERATE AN ERROR MESSAGE IF THIS IS NOT THE CASE.
;
; THIS SECTION OF CODE SHOULD BE INSERTED AFTER THE PROGRAM IS
; COMPLETED AT EACH LOCATION IN THE PROGRAM ENDING IN 75H.  IF THE
; PROGRAM IS EVER CHANGED TO VIOLATE THESE CONDITIONS, AN ERROR
; MESSAGE WILL BE GENERATED.
;
;=========================
;
IF      (LOW $) NE 075H ; CHECK FOR ADDRESS COUNTER ENDING IN 75H
        GARBAGE         ; GENERATE ERROR
ENDIF
        NOP
```

Fig. 2 - Code sequence which flags any use of location 75H for other than an NOP

```
;=========================
;
;
; INTERRUPT RETURN ROUTINE
;
; CHECKS THE STACK POINTER TO ENSURE THAT THE STACK
; IS LESS THAN MAXDEP DEEP.
;
; CHECKS RETURN ADDRESSES TO ENSURE THAT CONTROL
; RETURNS TO A PAGE WHICH CONTAINS LEGALLY INTERRUPTABLE
; CODE.
;
;=========================
;
INTRET:
MAXDEP EQU      2               ; MAXIMUM STACK DEPTH
       MOV      A,PSW
       DEC      A               ; POINT TO LAST LEVEL USED
       ANL      A,#00001110B    ; CHECK FOR VALIDITY OF DEPTH
       ADD      A,#(-MAXDEP)
       JC       RECOVR
       ADD      A,#MAXDEP       ; RESTORE MASKED PSW
       RL       A
       ADD      A,#09H          ; POINT TO TOP ENTRY IN STACK
       MOV      R1,A
       MOV      A,@R1
       ANL      A,#0FH          ; MASK OUT ALL BUT PAGE BITS
       ADD      A,#(-3)         ; PROHIBIT RETURNS TO PAGES 3-F(HEX)
       JC       RECOVR
;
;      RESTORE ACCUMULATOR AND MAKE NORMAL RETURN
;
       MOV      A,ASAVE
       EN       TCNTI
       RETR
```

Fig. 3 - Useful return from interrupt procedure

an action. This technique has been successfully employed in avionics applications requiring very high levels of fault tolerance.

The best defense against memory "hits" is to be aware that they will happen. Whenever possible, check limits of critical variables prior to and after computations are performed. For example, in controlling stepper motors assure that the desired position is indeed valid. Recalibrate absolute references (such as throttle reference point, closed-loop carburetor midpoint, etc.) whenever practical. If, for instance, the program requires a 3% actuator opening, it would be easy to recalibrate to 0 along the way.

If portions of a program are not time critical, the free time can be utilized to generate and update checksums in memory. In addition, it can verify key system variables that cannot be easily reconstructed without cold-starting the system.

Although time consuming, a procedure that has proved very effective and worthwhile is to check the validity of the stack pointer and return address prior to returning from a subroutine. This is not as difficult as it sounds. All subroutine returns can be centralized at the same point in the system, and instead of returning to the main program, each subroutine jumps to the "return" routine. This routine performs the following functions: First, it checks the stack pointer to verify that it is within a valid range, such as between 1 and 4 levels deep. Next, the return address is checked for existence (within a region of memory that contains executable code). Finally, (though somewhat overkilling the problem) the memory locations prior to the return address can be checked for the existence of a subroutine call opcode. Fig. 3 illustrates a useful return from interrupt procedure.

GRACEFUL RECOVERY

Once a system fault has been detected, the challenge remains to set the system back into correct operation without the cure being worse than the disease. A total cold start is rarely the optimal solution. A byproduct of good structured programming practice is that the program can easily be modified to leave tracks indicating how far along through various phases it has progressed. A typical microcomputer program for an engine control application will have several phases. The first will usually be "initialization", during which memory is put into a known state, counters are initialized, I/O is set into various modes of operation and execution started. A second phase of operation is "warm-up" during which the operation of the system changes as a function of time. This phase is usually included to allow components such as oxygen sensors to reach proper operating conditions. The third phase is "normal operation" during which the operation of the system is a function of the inputs, independent of elapsed time.

In general, a graceful recovery to normal operation should bypass phase 2 altogether, and as much of phase 1 as possible. One technique for determining how badly the processor has been "hit" is to initialize otherwise unused memory locations with known, psuedo-random patterns (avoid all ones or zeros), and checking them as part of the recovery routine. If these test locations have been affected, then chances are other bytes of memory have been "hit" and all of phase 1 will be necessary. An example is shown in Fig. 4. Whether or not phase 2 should be

```
;=========================
;
;
; HOTCLD - COMPARES 4 MEMORY LOCATIONS WITH A PSEUDO-RANDOM
; BIT PATTERN.  IF THERE IS A DISCREPANCY, THE MEMORY IS
; RE-INITIALIZED AND CONTROL PASSES TO A COLD START ROUTINE.
; IF MEMORY IS STILL VALID, CONTROL PASSES TO A WARM START
; ROUTINE.
;
;=========================
;
HOTCLD:
       CLR      F0      ; USE FLAG 0 AS WARM/COLD FLAG
       CPL      F0      ; INITIALIZE TO 1
       MOV      R0,#(LOW RANDNO)        ; PNTR0 = ADDRESS OF RANDOM NOS.
       MOV      R1,BITTBL               ; RAM LOCATION OF RANDOM NOS.
       MOV      R2,#4   ; COUNTER FOR 4 VALUES
HOTC1:
       MOV      A,R0
       MOVP3    A,@A    ; GET RANDOM NUMBER
       XCH      A,@R1   ; RANDOM NUMBER IN RAM
       XRL      A,@R1   ; COMPARE CORRECT VALUE WITH PREVIOUS RAM VALUE
       JZ       HOTC2
       CLR      F0      ; INDICATES AN ERROR
HOTC2:
       INC      R0      ; NEXT RANDOM NUMBER
       INC      R1      ; NEXT RAM LOCATION
       DJNZ     R2,HOTC1        ; REPEAT IF NOT DONE
       JF0      COLD    ; COLD START IF ERROR DETECTED
       JMP      WARM
;
; RANDOM NUMBER TABLE IN PAGE 3 OF MEMORY
;
RANDNO:
       DB       10010101B
       DB       11001100B
       DB       01111010B
       DB       00100101B
       NOP
       JMP      RECOVR
```

Fig. 4 - Graceful recovery

```
;=========================
;
;
; RECOVR ROUTINE - SOFTWARE RESET
;
; JUMPED TO WHENEVER THE PROGRAM DETERMINES THAT A HARDWARE
; FAILURE HAS OCCURRED.
;
; USES SOFTWARE TO SET UP INITIAL CONDITONS LIKE THOSE
; CAUSED BY A HARDWARE RESET.
;
;=========================
;
RECOVR:
        DIS     TCNTI
        DIS     I       ; DISABLE ALL INTERRUPTS
        CLR     A       ; CLEAR FLAGS
        MOV     PSW,A
        SEL     MB0     ; BANKS 0 FOR REGISTERS AND MEMORY
        SEL     RB0
        MOVX    A,@R0   ; THREE-STATE BUS FOR INPUT
        JMP     COLD    ; JUMP TO APPROPRIATE COLD OR WARM START
```

Fig. 5 - Typical 8048 recovery routine

executed is a difficult issue. Since memory has been seriously affected, any flags that indicate whether phase 3 has been entered are not necessarily valid, and a tradeoff must be made toward the lesser of two evils; that is, whether an extraneous warm-up is worse than no warm-up.

For many of the previously mentioned failure modes, selective recovery schemes can be utilized. Again, tradeoffs must be made depending on the details of the system. The ultimate goal of any recovery scheme is to place the processor in a known, correct state. A typical 8048 recovery routine is illustrated in Fig. 5.

FILTERING

A time-honored tenet of computing is "garbage in, garbage out". However, a system is often expected to correctly process the garbage. While a noisy input will not cause a lock-up condition, it can certainly cause incorrect results, and a possible loss of synchronization between the program and external variable conditions. Programs can be de-sensitized to noisy inputs by various filtering algorithms. All input filtering algorithms imply a tradeoff betwen input bandwidth and noise immunity. A convenient algorithm for sampling input pins with any 8-bit microcomputer is to dedicate a byte of memory for each input and rotate the pin value into the byte at clocked intervals, usually in the millisecond range for switch inputs. Whenever the byte contains all ones, the pin is treated as a one. Whenever the byte contains all zeros, the pin is assumed to be 0. Otherwise it is assumed that the input has not changed since the last time one of these conditions was met. The sampling rate times 8 thus becomes the response time penalty, and so the sampling rate must be set accordingly.

HARDWARE TO THE RESCUE

The use of a dead-man timer has been previously mentioned. The purpose of a dead-man timer is to place the processor into a known state at particular intervals of time, unless periodically defeated by the software (it kicks you, unless you keep kicking it!) It should be reasonably difficult to accidentally defeat the timer to reduce the chance of occurance while the processor is stuck inside a "tight loop". In typical systems, the dead-man timer generates a hardware reset to place the processor into a known state.

Where the processor contains "one-way" opcodes, it is often useful for the program to be able to generate a hardware reset through external circuitry in order to cancel those operations. This can be accomplished by connecting an otherwise unused control strobe to discharge the reset capacitor. This presents a cost/reliability tradeoff problem.

EXPERIMENTAL RESULTS

Lest the reader be discouraged by all the pitfalls of designing systems to operate in noisy environments, it is worthwhile to discuss experimental results. In a particular design, an Intel 8048 was programmed using structured techniques and placed in proximity to a 1/4" spark gap powered by a typical automotive ignition system. At a distance of several inches from the spark gap, the processor would occasionally lock up. At this distance, several hundred millivolts of noise was measured on all the 8048 pins including power supplies. After the techniques described in this paper were applied to the code, the processor could operate at a distance of less than 1 inch to the spark gap without lock up. At this distance, as much as 5 volts of noise was measured on various pins. To be sure, erratic operation followed virtually every spark, but in each case, graceful recovery was possible when the spark was removed.

SUMMARY

Execution speed and memory size can be sacrificed in many systems to enhance the noise immunity of the basic algorithms. Within practical limits, most microcomputer programs can be made fault-tolerant to soft failures in the hardware environment. While the faults can never be totally eliminated, the impact can be reduced. Software for noisy environments should be able to detect and correct various lock-up and soft error conditions.

ACKNOWLEDGEMENTS

Dan Tsiang and Ira Russell - Ford Motor Company.

REFERENCES

1. "MCS-48 Microcomputer User's Manual" Intel Corporation, 1977.
2. "MCS-48 Assembly Language Reference Card" Intel Corporation, 1976.
3. Dr. L. A. Leventhal, "Structured Programming Formulates Microprocessor Program Logic". Digital Design, October 1978